Text Compression

A paradox of modern computer technology is that despite an almost incomprehensible increase in storage and transmission capacities, more and more effort has been put into using compression to increase the amount of data that can be handled. No matter how much storage space or transmission bandwidth is available, someone always finds something to fill it with. It seems that Parkinson's law applies to space as well as time.

The problem of representing information efficiently is nothing new. People have always been interested in means for storing and communicating information, and methods for compressing text to improve this process predate computers. For example, the Braille code for the blind can include "contractions," which represent common words with two or three characters, and Morse code also "compresses" data by using shorter representations for common characters.

Text compression on a computer involves changing the representation of a file so that it takes less space to store or less time to transmit, yet the original file can be reconstructed exactly from the compressed representation. *Text* compression techniques are distinguished from the more general *data* compression methods because the original file can always be reconstructed exactly. For some types of data other than text, such as sound or images, small changes, or *noise*, in the reconstructed data can be tolerated because it is a digital approximation to an analog waveform anyway. However, with text it must be possible to reproduce the original file exactly.

Many compression methods have been invented and reinvented over the years. These range from numerous ad hoc techniques to more principled methods that can give very good compression. One of the earliest and best-known methods of text compression for computer storage and telecommunications is Huffman coding.[1]

---

1 David Huffman, then a student at M.I.T., devised his celebrated coding method in response to a challenge from his professor, and as a result managed to avoid having to take the final exam for the course!

This uses the same principle as Morse code: common symbols—conventionally, characters—are coded in just a few bits, while rare ones have longer codewords. First published in the early 1950s, Huffman coding was regarded as one of the best methods of compression for several decades, until two breakthroughs in the late 1970s—Ziv-Lempel compression and arithmetic coding—made higher compression rates possible. Both these ideas achieve their power through the use of *adaptive compression*—a kind of dynamic coding where the input is compressed relative to a model that is constructed from the text that has just been coded. By basing the model on what has been seen so far, adaptive compression methods combine two key virtues: they are able to encode in a single pass through the input file, and they are able to compress a wide variety of inputs effectively rather than being fine-tuned for one particular type of data such as English text.

Ziv-Lempel methods are adaptive compression techniques that give good compression yet are generally very fast and do not require large amounts of memory. The idea behind them was developed by two Israeli researchers, Jacob Ziv and Abraham Lempel, in the late 1970s. *Arithmetic coding* is really an enabling technology that makes a whole class of adaptive compression schemes feasible, rather than a compression method in its own right. Early implementations of character-level Huffman coding were typically able to compress English text to about five bits per character. Ziv-Lempel methods reduced this to fewer than four bits per character—about half the original size. Methods based on arithmetic coding further improved the compression to just over two bits per character. The price paid is slower compression and decompression, and more memory required in the machines that do the processing.

Some of the best compression methods available are variants of a technique called *prediction by partial matching* (PPM), which was developed in the early 1980s. PPM relies on arithmetic coding to obtain good compression performance. Since then, there has been little advance in the amount of compression that can be achieved, other than some fine-tuning of the basic methods, and the development of a new method called *block sorting* that gives similar performance to PPM. On the other hand, many techniques have been discovered that improve the speed or memory requirements of compression methods. Most of these achieve a significant reduction in computing requirements in exchange for a slight loss of compression.

Present compression techniques give compression of about two bits per character for general English text, depending on what you mean by "general English text." Evidence suggests that compression better than one bit per character is not likely to be achieved, and in order to approach this bound, compression methods will have to draw both on the semantic content of the text and external world knowledge. This is discussed further in Section 2.8.

Improvements are still being made in processor and memory utilization during compression, although both of these resources are becoming cheaper and more plentiful. Generally speaking, the amount of compression achieved by the PPM method increases as more memory becomes available. It is not competitive with Ziv-Lempel methods until 100 Kbytes or more are available, and it does not approach its best performance until 500 Kbytes to 1 Mbyte is allocated. Because of

this requirement, when PPM was first proposed in the early 1980s it was a laboratory curiosity, requiring a large minicomputer to test it. Now, most PCs have sufficient computing power to execute it quite effectively. Furthermore, processor speed is currently improving at a faster rate than disk speeds and capacities. Since compression decreases the demand on storage devices at the expense of processing, it is becoming more economical to store data in a compressed form than uncompressed.

Most text compression methods can be placed in one of two classes: *symbolwise* methods and *dictionary* methods. Symbolwise methods work by estimating the probabilities of symbols (often characters), coding one symbol at a time, using shorter codewords for the most likely symbols in the same way that Morse code does. Dictionary methods achieve compression by replacing words and other fragments of text with an index to an entry in a "dictionary." The Braille code is a dictionary method since special codes are used to represent whole words.

Symbolwise methods are usually based on either Huffman coding or arithmetic coding, and they differ mainly in how they estimate probabilities for symbols. The more accurately these estimates are made, the greater the compression that can be achieved. To obtain good compression, the probability estimate is usually based on the context in which a symbol occurs. The business of estimating probabilities is called *modeling*, and good modeling is crucial to obtaining good compression. Converting the probabilities into a bitstream for transmission is called *coding*. Coding is well understood and can be performed very effectively using either Huffman coding or arithmetic coding. Modeling is more of an art, and there does not appear to be any single "best" method.

Dictionary methods generally use quite simple representations to code references to entries in the dictionary. They obtain compression by representing several symbols as one output codeword. This contrasts with symbolwise methods, which rely on generating good probability estimates for a symbol, since the length of the output codeword is what determines compression performance; for this reason, symbolwise methods are sometimes referred to as *statistical* methods, since they rely on estimating accurate statistics. The most significant dictionary methods are based on Ziv-Lempel coding, which uses the idea of replacing strings of characters with a reference to a previous occurrence of the string. This approach is adaptive, and it is effective because most characters can be coded as part of a string that has occurred earlier in the text. Compression is achieved if the reference, or *pointer*, is stored in fewer bits than the string it replaces. There are many variations on Ziv-Lempel coding, with different pointer representations and different rules governing which strings can be referenced.

The key distinction between symbolwise and dictionary methods is that symbolwise methods generally base the coding of a symbol on the context in which it occurs, whereas dictionary methods group symbols together, creating a kind of implicit context. Hybrid schemes are possible, in which a group of symbols is coded together and the coding is based on the context in which the group occurs. This does not necessarily provide better compression than symbolwise methods, but it can improve the speed of compression.

The following sections describe in more detail the main compression techniques introduced above. We look at the modeling and coding components separately. First the general idea of modeling is introduced, and then a particularly powerful class of models, adaptive models, is discussed. Before looking at how models are used in practice, we describe the two principal methods of coding used to represent symbols based on the probability distributions generated by models. Each of these descriptions begins with an overview of what the method is and how it works, and each is followed by a much more detailed description of how the coding can be implemented efficiently. These details are included because, although they lead to extremely effective implementations, they are not obvious and are hard to come by in the literature; they should be skipped on a first reading. The two main classes of models, symbolwise and dictionary, are then examined. There is a section that deals with the problem of providing random access to compressed text, which is important for full-text retrieval systems. Finally, the practical performance of various text compression methods is discussed.

2.1 Models

Compression methods obtain high compression by forming good models of the data that is to be coded. The function of a model is to *predict* symbols, which amounts to providing a probability distribution for the next symbol that is to be coded. For example, during the encoding of a text, the "prediction" for the next symbol might include a probability of 2 percent for the letter *u*, based on its relative frequency in a sample of text. The set of all possible symbols is called the *alphabet*, and the probability distribution provides an estimated probability for each symbol in the alphabet.

The model provides this probability distribution to the encoder, which uses it to encode the symbol that actually occurs. The decoder uses an identical model together with the output of the encoder to find out what the encoded symbol was. Figure 2.1 illustrates the whole process.

The number of bits in which a symbol, s, should be coded is called the *information content* of the symbol. The information content, $I(s)$, is directly related to the symbol's predicted probability, $\Pr[s]$, by the function $I(s) = -\log \Pr[s]$ bits.[2] For example, to transmit a symbol representing the fact that the outcome of a fair coin toss was "heads," the best an encoder can do is to use $-\log(1/2) = 1$ bit. The average amount of information per symbol over the whole alphabet is known as the *entropy* of the probability distribution, denoted by H. It is given by

$$H = \sum_s \Pr[s] \cdot I(s) = \sum_s -\Pr[s] \cdot \log \Pr[s].$$

2 All logarithms in this book are to base two unless indicated otherwise.

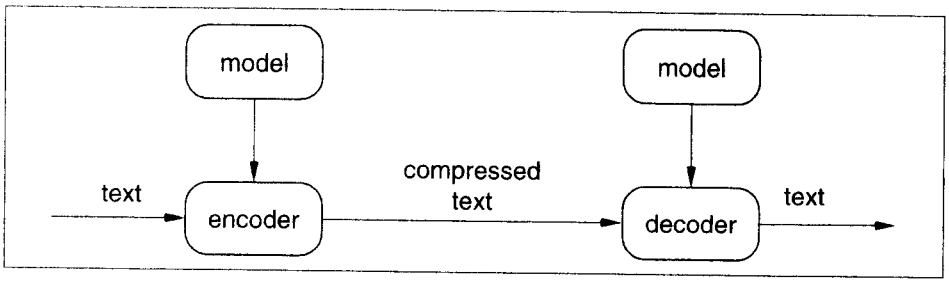

Figure 2.1 Using a model to compress text.

Provided that the symbols appear independently and with the assumed probabilities, H is a *lower* bound on compression, measured in bits per symbol, that can be achieved by *any* coding method. This is the celebrated source coding theorem of Claude Shannon, a Bell Labs scientist who single-handedly developed the field of information theory, and provides a bound that we can strive to attain but can never beat (Shannon 1948).

Huffman coding often achieves compression performance close to the entropy, but can, in some cases, be very inefficient. One such situation is when very good predictions are being made, in which case probabilities close to one are generated. This is exactly when entropy is minimized and "compressibility" is maximized and is what we hope to achieve when designing data compression systems; hence it is unfortunate that Huffman coding is inefficient in this situation. By way of contrast, the more recent method of arithmetic coding comes arbitrarily close to the entropy even when probabilities are close to one and the entropy of the probability distribution is close to zero. These two methods are discussed in Sections 2.3 and 2.4. What is important for the model is to provide a probability distribution that makes the probability of the symbol that actually occurs as high as possible. The above relationship means that a low probability results in a high entropy and vice versa. In the extreme case when $\Pr[s] = 1$, only one symbol s is possible, and $I(s) = 0$ indicates that zero bits are needed to transmit it. This follows intuitively: if a symbol is certain to occur, then it conveys no information and need not be transmitted. To paraphrase a well-worn truism, $I(\text{death}) = 0$ and $I(\text{taxes}) = 0$.

Conversely, I becomes arbitrarily large as $\Pr[s]$ approaches zero, and a symbol with zero probability cannot be coded. In practice, all symbols must be given a nonzero probability because a zero-probability symbol could not be coded if, by unlucky chance, it did occur. Moreover, it is not possible for the encoder to peek at the next symbol and artificially boost its probability just for this step; the encoder and decoder must use the *same* probability distribution, and the decoder clearly cannot look ahead at symbols that have not yet been decoded. Hence the model must take into account all the information available to the decoder and then gamble on what the next symbol will be. The best compression is obtained when the model is backing the symbols that actually occur.

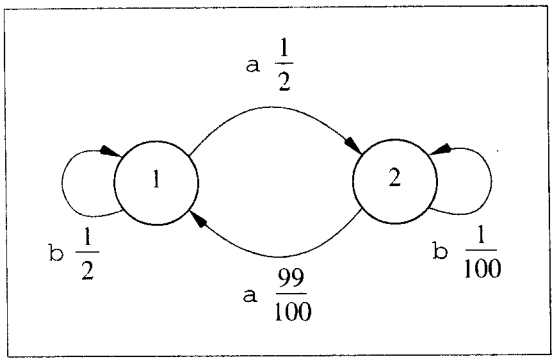

Figure 2.2 A simple finite-state model.

At the beginning of this section the probability of a u was estimated as 2 percent. This corresponds to an information content of 5.6 bits; that is, if it does happen to be the next symbol, u should be transmitted in 5.6 bits. Nothing has been said yet about *how* the probabilities should be estimated. It turns out that predictions can usually be improved by taking account of the previous symbol. If a q has just been encountered, the probability of u may jump to 95 percent, based on how often q is followed by u in a sample of text. This gives a much lower information content for u of 0.074 bits. Of course, other symbols must have lower probabilities and therefore longer codewords to compensate, and if the prediction is incorrect (as in *Iraq* or *Qantas*), the price paid is extra bits in the output. But averaged over many appearances of the context q, the number of bits required to decode each appearance of u can be expected to decrease.

Models that take a few immediately preceding symbols into account to make a prediction are called *finite-context* models of order m, where m is the number of previous symbols used to make the prediction. Such models are effective in a variety of compression applications, and the best text compression methods known are based on this approach.

Other approaches to modeling are possible, and although potentially more powerful, they have not proved as popular as finite-context models. One approach is to use a *finite-state* model, in which each state of a finite-state machine stores a different probability distribution for the next symbol. Figure 2.2 shows such a model. This particular model is for strings in which the symbol a is expected to occur in pairs. Encoding starts in state 1, where a and b are predicted with equal probability, $1/2$. Using the formula above, we find that each should be coded in one bit (not surprisingly). If a b is received, the encoder stays in state 1 and uses the same probability distribution for the next symbol. However, if an a is received, it moves to state 2, where the probability of a b is now only $1/100$ and requires 6.6 bits to be encoded, as opposed to 0.014 bits to encode an a. This model captures behavior that cannot be represented accurately by a finite-context model because a state model is able to keep track of whether an odd or even number of as have occurred consecutively.

It is important that the decoder works with an identical probability distribution in order to decode symbols correctly. This is achieved by ensuring that it has an identical model to the encoder's and that it starts in the same state as the encoder. The encoder transmits the symbol and then follows a transition; the decoder recovers the symbol and can then follow the same transition, so it is now in the same state as the encoder and will use the same probability distribution for the next symbol. Error-free transmission is assumed, for if any errors were to occur, the encoder and decoder would lose synchronization, with potentially catastrophic results.

If the text being compressed is in a formal language such as C or Java, a grammar can be used to model the language. The text is represented by sending the sequence of *productions*, or rules, that would generate the text from the grammar. By estimating the probability of a particular production occurring, more frequently used productions can be coded in fewer bits, thus achieving good compression. It is hard to obtain a formal grammar for texts written in natural languages, so to date, grammar models have been applied only to artificial languages such as programming languages.

2.2 Adaptive models

There are many ways to estimate the probabilities in a model. We could conceivably guess suitable probabilities when setting up a compression system and use the same distribution for all input texts. However, it is easier, and more accurate, to estimate the probabilities from a sample of the kind of text that is being encoded.

The method that always uses the same model regardless of what text is being coded is called *static* modeling. Clearly, this runs the risk of receiving an input that is quite different from the one for which the model was set up—for example, a model for the English language will probably not perform well with a file of numbers and vice versa. One example of such a mismatch occurs when numeric data is transmitted using Morse code. Because the digits are all relatively rare in normal text, they are assigned long codewords, and so transmission times increase if documents such as financial statements are sent. Another example is shown in Figure 2.3, which is the opening sentence of a rather contorted book—*Gadsby* by E. V. Wright, published in 1939. You may care to try to work out what is unusual about the text before reading on. In fact, the reason that the text reads strangely is that it does not contain a single occurrence of what is usually the most common letter in normal English text—*e*. A static model designed for normal English text would perform poorly in this case.

One solution is to generate a model specifically for each file that is to be compressed. An initial pass is made through the file to estimate symbol probabilities, and these are transmitted to the decoder before transmitting the encoded symbols. This approach is called *semi-static* modeling. (Semi-static modeling has also been referred to as semi-adaptive modeling, but we prefer the term "semi-static" because the implementation of these models has more in common with static models than adaptive ones.) Semi-static modeling has the advantage that the model is invariably

```
If Youth, throughout all history, had had a champion to stand
up for it; to show a doubting world that a child can think;
and, possibly, do it practically; you wouldn't constantly run
across folks today who claim that ''a child don't know anything.''
```

Figure 2.3 The first sentence of an unusual book.

```
"I never heerd a skilful old married feller of twenty
years' standing pipe ''my wife'' in a more used note
than 'a did," said Jacob Smallbury.  "It migh
```

Figure 2.4 Sample text.

better suited to the input than a static one, but the penalty paid is having to transmit the model first, as well as the preliminary pass over the data to accumulate symbol probabilities. In some situations, such as interactive data communications, it may be impractical to make two passes over the data, and for complex models the cost of pretransmitting the model might be a considerable overhead.

Adaptive modeling is an elegant solution to these problems. An adaptive model begins with a bland probability distribution and gradually alters it as more symbols are encountered. As an example, consider an adaptive model that uses the previously encoded part of a string as a sample to estimate probabilities. We will use a model that operates character by character, with no context used to predict the next symbol—in other words, each character of the input is treated as an independent symbol. Technically, this is called a *zero-order* (equivalently, *order-0*) model: in full, an adaptive, zero-order, character-level model. Now consider the text of Figure 2.4, excerpted from Thomas Hardy's book *Far from the Madding Crowd*. It is from the final scene in the book, in which a group is jesting with a newlywed couple. The archaic language in this excerpt is another reminder of the desirability of using adaptive codes.

The zero-order probability that the next character after the excerpt is t is estimated to be 49,983/768,078 = 6.5 percent, since in the previous text, 49,983 of the 768,078 characters were ts. Using the same system, an e has probability 9.4 percent, and an x has probability 0.11 percent. The model provides this estimated probability distribution to an encoder such as an arithmetic coder (see Section 2.4). In fact, the next character is a t, which an arithmetic coder represents in about $-\log 0.065 = 3.94$ bits. The decoder is able to generate the same model since it has just decoded all the characters up to (but not including) the t. It makes the same probability estimates as the encoder and so is able to decode the t correctly when it is received. In practice, the encoder and decoder do not extract the statistics from the prior text each time they are needed, but instead keep a running tally of the character counts.

Some details of the adaptive system need to be considered. First, the system must avoid the situation in which a character is predicted with a probability of zero. In the example above, the character Z has never occurred in the text up to this point and would be predicted with zero probability. Such events cannot be coded, yet they might occur; this is referred to as the *zero-frequency problem* (Witten and Bell 1991). The text *It mighZ* is very unlikely, but it can occur. If nowhere else, it has just occurred in this book.

There are several ways to solve the zero-frequency problem. One is to allow one extra count, which is divided evenly among any symbols that have not been observed in the input. In the Hardy example, the total count would be increased by one to 768,079. A total of 82 different characters have been seen so far, so 46 of the 128 ASCII characters have not occurred by this point. Each of these gets $1/46$ of the spare proportion of $1/768{,}079$. Thus, a Z character is given a probability of $1/(46 \times 768{,}079) = 1/35{,}331{,}634$, corresponding to 25.07 bits in the output.

Another possibility is to artificially inflate the count of every character in the alphabet by one, thereby ensuring that none has a zero frequency. This is equivalent to starting the model assuming that we have already processed a stretch of text in which each possible character appeared exactly once. In the above example, allowing for the ASCII alphabet of 128 characters, 128 would be added to the total number of characters seen so far, giving Z a relative frequency of $1/768{,}206 = 0.00013$ percent, corresponding to 19.6 bits in the output.

Several other solutions to the zero-frequency problem are possible, although in general none offers a particularly significant compression performance advantage over the others. The problem is most acute near the beginning of a text where there are few, if any, samples on which to base estimates; so at face value the choice of method is more critical for small texts than for large ones. The method is also important for models that use very many different contexts because many of the contexts will be used only a few times.

The example above used a zero-order model, in which each character's probability was estimated without regard to context. For a higher-order model, such as a first-order model, the probability is estimated by how often that character has occurred in the current context. For example, the excerpt used above to illustrate a zero-order model was coding the letter t in the context of the phrase *It migh*, but in reality made no use at all of the characters comprising that phrase. On the other hand, a first-order model would use the final h as a context with which to *condition* the probability estimates. The letter h has occurred 37,525 times in the prior text, and 1,133 of these times it was followed by a t. Ignoring for a moment the zero-frequency problem, the probability of a t occurring after an h can be estimated to be $1{,}133/37{,}525 = 3.02$ percent, which would have it coded in 5.05 bits. This is actually worse than the zero-order estimate because the letter t is rare in this context—an h is much more likely to be followed by an e, and so here is an example where use of more information caused inferior compression. On the other hand, a second-order model does substantially better. It uses the relative frequency that the context gh is followed by a t, which is 1,129 times out of 1,754, or 64.4 percent, and results in the t being coded in just 0.636 bits.

So far we have suggested how the probabilities in a model can be adapted, but it is also possible—and effective—to adapt a model's *structure*. In a finite-context model, the structure determines which contexts are used; in a finite-state model, the structure is the set of states and transitions available. Adaptation usually involves adding more detail to an area of the model that is heavily used. For example, if the first-order context *h* is being used frequently, it might be worthwhile to add more specific contexts, such as the second-order contexts *th* and *sh*. So long as the encoder and decoder use the same rules for adding contexts, and the decision to add contexts is based on the previously encoded text only, they will remain synchronized.

Adaptive modeling is a powerful tool for compression and is the basis of many successful methods. It is robust, reliable, and flexible. The principal disadvantage is that it is not suitable for random access to files—a text can be decoded only from the beginning, since the model used for coding a particular part of the text is determined from all the preceding text. Hence, adaptive modeling is ideal for general-purpose compression utilities but is not necessarily appropriate for full-text retrieval. This point will be taken up again in Section 2.7.

2.3 Huffman coding

Coding is the task of determining the output representation of a symbol, based on a probability distribution supplied by a model. The general idea is that a coder should output short codewords for likely symbols and long codewords for rare ones. There are theoretical limits on how short the average length of a codeword can be for a given probability distribution, and much effort has been put into finding coders that achieve this limit. Another important consideration is the speed of the coder—a reasonable amount of computation is required to generate near-optimal codes. If speed is important, we might use a coder that sacrifices compression performance to reduce the amount of effort required. For example, if there are 256 possible symbols to be coded, we might use a coder that represents the 15 most probable symbols in 4 bits and the remainder in 12 bits. The extreme of this sort of approximation is just to code all symbols in 8 bits. It gives no compression but is very fast. In fact, many dictionary-based methods use a simple coder like this, with the implicit assumption that the symbols (which in this kind of model are actually groups of characters) are equally likely.

In contrast to dictionary methods, symbolwise schemes depend heavily on a good coder to achieve compression, and most research on coders has been performed with symbolwise methods in mind. This section and the next describe the two main methods of coding: *Huffman coding* and *arithmetic coding*. Huffman coding tends to be faster than arithmetic coding, but arithmetic coding is capable of yielding compression that is close to optimal given the probability distribution supplied by the model. For each of these two types of coder, we first look at the principle by which they achieve compression and then give details of how they are implemented in practice. We begin with Huffman coding (Huffman 1952).

Table 2.1 Codewords and probabilities for a seven-symbol alphabet.

Symbol	Codeword	Probability
a	0000	0.05
b	0001	0.05
c	001	0.1
d	01	0.2
e	10	0.3
f	110	0.2
g	111	0.1

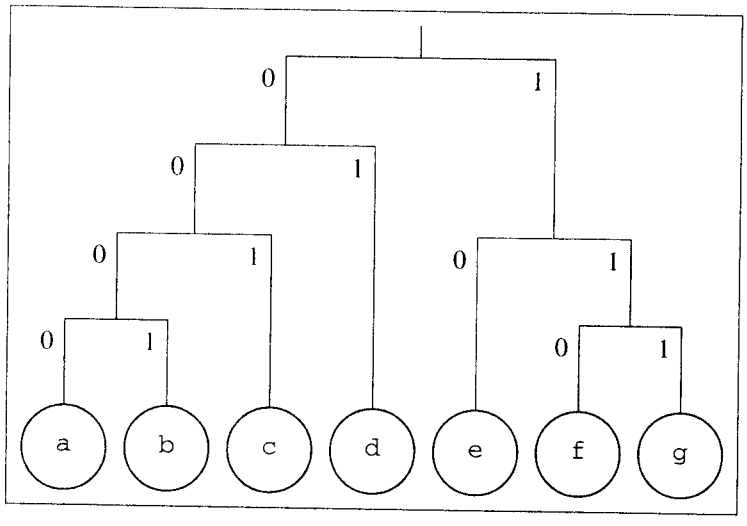

Figure 2.5 A Huffman code tree.

Table 2.1 shows codewords for the seven-symbol alphabet a, b, c, d, e, f, and g. A phrase is coded by replacing each of its symbols with the codeword given by the table. For example, the phrase *eefggfed* is coded as 10101101111111101001. Decoding is performed from left to right. The input to the decoder begins with 10 . . . , and the only codeword that begins with this is the one for e, which is therefore taken as the first symbol. Decoding then proceeds with the remainder of the string, 1011011 . . . .

Figure 2.5 shows a tree that can be used for decoding. The tree is traversed by starting at the root and following the branch corresponding to the next bit in the coded text. The path from the root to each symbol (at a leaf) corresponds to the codewords in Table 2.1. This type of code is called a *prefix* code—or more accurately, a *prefix-free* code—because no codeword is the prefix of another symbol's codeword.

If that were not the case, the decoding tree would have symbols at internal nodes, which leads to ambiguity in decoding.

The code in Table 2.1 was produced by the technique of Huffman coding, which generates codewords for a set of symbols, given some probability distribution for the symbols. The codewords generated yield the best compression possible for a prefix-free code for the given probability distribution.

Huffman's algorithm works by constructing the decoding tree from the bottom up. For the example symbol set, with the probabilities shown in Table 2.1, it starts by creating for each symbol a leaf node containing the symbol and its probability (Figure 2.6a). Then the two nodes with the smallest probabilities become siblings under a parent node, which is given a probability equal to the sum of its two children's probabilities (Figure 2.6b).

The combining operation is repeated, choosing the two nodes with the smallest probabilities and ignoring nodes that are already children. For example, at the next step the new node formed by combining a and b is joined with the node for c to make a new node with probability $p = 0.2$. The process continues until there is only one node without a parent, which becomes the root of the decoding tree (Figure 2.6c). The two branches from every nonleaf node are then labeled 0 and 1 (the order is not important) to form the tree.

Figure 2.7 shows the general algorithm for constructing a Huffman code. The algorithm is expressed in terms of a set T that recursively contains other sets, with each subset corresponding to a node in the tree. When the algorithm terminates, T contains one set, which itself contains two sets—the descriptions of the two subtrees of the root. A more detailed description of how Huffman coding is implemented appears later in this section.

Huffman coding is generally fast for both encoding and decoding, provided that the probability distribution is static. There are also algorithms for adaptive Huffman coding, where localized adjustments are made to the tree to maintain the correct structure as the probabilities change (Gallager 1978; Cormack and Horspool 1984; Knuth 1985; Vitter 1989). However, the better adaptive symbolwise models usually use many different probability distributions at the same time, with the appropriate distribution being chosen depending on the context of the symbol being coded. Huffman coding requires that multiple trees be maintained in this situation, which can become demanding on memory. The alternative is for each tree to be regenerated whenever it is required, but this is slow. Hence, for adaptive compression, arithmetic coding (described in the next section) is usually preferable, as its speed is comparable to that of adaptive Huffman coding, yet it requires less memory and is able to achieve better compression—particularly when high-probability events are being coded.

Nevertheless, Huffman coding turns out to be very useful for some applications. For example, when coupled with a word-based (rather than character-based) model, it gives good compression, and its speed and ease of random access make it more attractive than arithmetic coding. Furthermore, there is a slightly different representation of a Huffman code that decodes very efficiently despite the extremely large models that might arise with a word-based model. This representation is called the

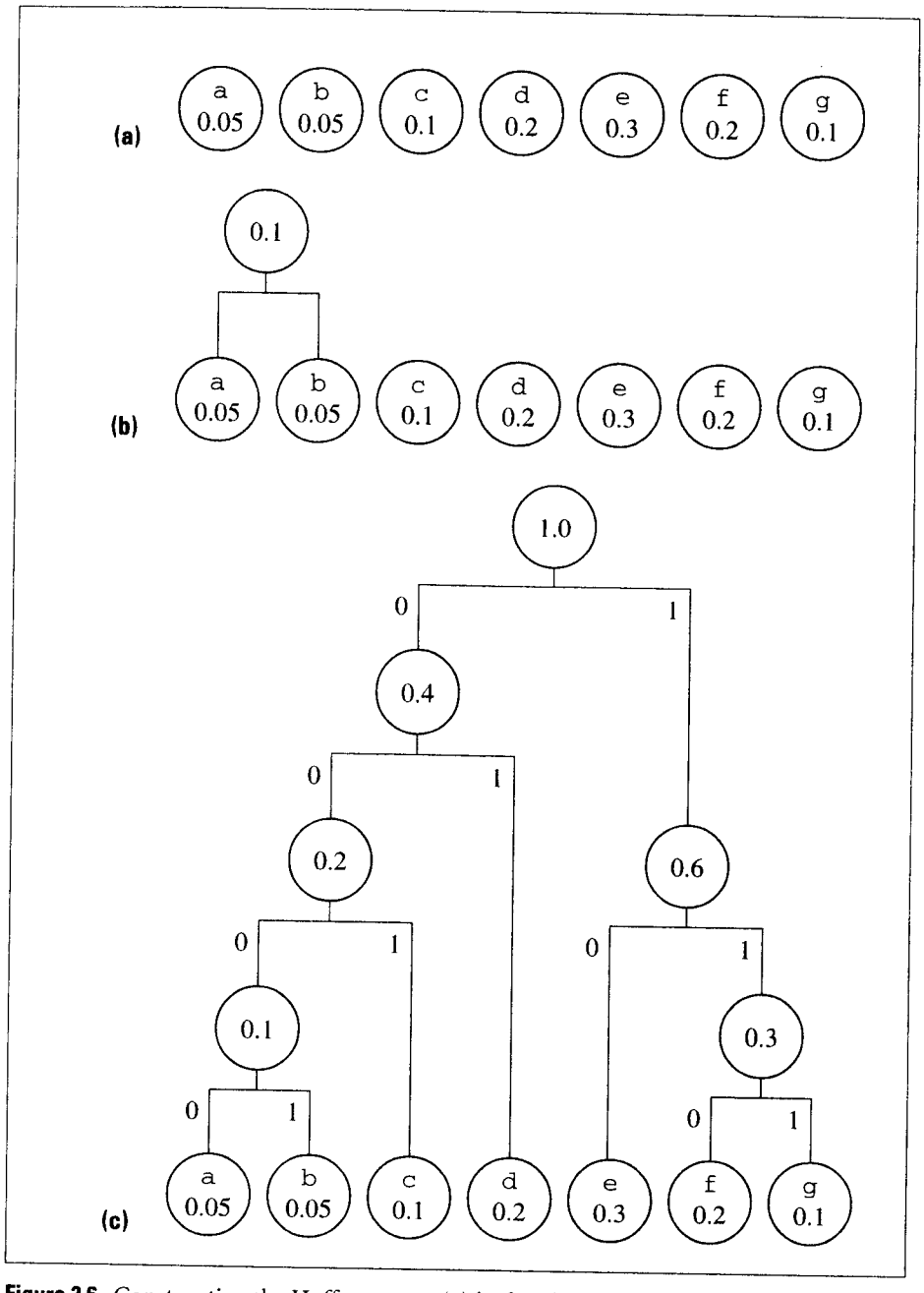

Figure 2.6 Constructing the Huffman tree: (a) leaf nodes; (b) combining nodes; (c) the finished Huffman tree.

To calculate a Huffman code,

1. Set $T \leftarrow$ a set of n singleton sets, each containing one of the n symbols and its probability.
2. Repeat $n - 1$ times
   (a) Set m_1 and $m_2 \leftarrow$ the two subsets of least probability in T.
   (b) Replace m_1 and m_2 with a set $\{m_1, m_2\}$ whose probability is the sum of that of m_1 and m_2.
3. T now contains only one item, which corresponds to the root of a Huffman tree; the length of the codeword for each symbol is given by the number of times it was joined with another set.

Figure 2.7 Assigning a Huffman code.

canonical Huffman code (Hirschberg and Lelewer 1990). It uses the same codeword *lengths* as a Huffman code, but imposes a particular choice on the codeword *bits*.

Table 2.2 shows part of a canonical Huffman code for the Hardy book, where the alphabet has been chosen to be the words that appear in the book. The frequency of each word has been counted, and, as in the conventional Huffman method, the codewords have been chosen to minimize the size of the compressed file for this model. In the terminology introduced on page 28, this is a static zero-order word-level model. The codewords are shown in decreasing order of length, and therefore in increasing order of word frequency—except that within each block of codes of the same length, words are ordered alphabetically rather than by frequency. The list begins with the thousands of words (and numbers) that appear only once. Words that occur only once in a text are called *hapax legomena*, a term that we will meet again on several occasions. Many of the words, such as *yopur* and *youmg*, occur only once because they are typographical errors. The numbers 100, 101, ... come from page numbers that are recorded in the file. (They start at 100 rather than a smaller number because words of the same codeword length are sorted in lexical—not numerical—order, so that 90, 91, ... appear later in the sequence.)

The table shows the codewords sorted from longest to shortest. An important feature of a canonical code like this is that when the codewords are sorted in lexical order—that is, when they are in the sequence they would be in if they were entries in a dictionary—they are also in order from the longest to the shortest codeword. On the other hand the code of Table 2.1 does not exhibit this property: although the codewords are ordered lexicographically, this does not result in them being sorted by length.

The key to using canonical codes efficiently is to notice that a word's encoding can be determined quickly from the length of its codeword, how far through the list it is, and the codeword for the first word of that length. For example, the word *said* is the 10th seven-bit codeword. Given this information and that the first seven-bit

Table 2.2 A canonical Huffman code.

Symbol	Codeword	
	Length	Bits
100	17	00000000000000000
101	17	00000000000000001
102	17	00000000000000010
103	17	00000000000000011
...	...	...
yopur	17	00001101010100100
youmg	17	00001101010100101
youthful	17	00001101010100110
zeed	17	00001101010100111
zephyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
...	...	...
were	8	10100110
which	8	10100111
as	7	1010100
at	7	1010101
for	7	1010110
had	7	1010111
he	7	1011000
her	7	1011001
his	7	1011010
it	7	1011011
s	7	1011100
said	7	1011101
she	7	1011110
that	7	1011111
with	7	1100000
you	7	1100001
I	6	110001
in	6	110010
was	6	110011
a	5	11010
and	5	11011
of	5	11100
to	5	11101
the	4	1111

codeword is 1010100, we can obtain the codeword for *said* by incrementing 1010100 nine times, or, more efficiently, adding nine to its binary representation.

Canonical codes come into their own for decoding because it is not necessary to store a decode tree. All that is required is a list of the symbols ordered according to the lexical order of the codewords, plus an array storing the first codeword of each distinct length. For example, with the code in Table 2.2, if the upcoming bits to be decoded are 1100000101 . . . , then the decoder will quickly determine that the next codeword must come after the first seven-bit word, *as* (1010100), and before the first six-bit word, *I* (110001). Therefore, the next seven bits are read (1100000), and the difference of this binary value from the first seven-bit value is calculated. In this example, the difference is 12, which means that the word is 12 positions after the word *as* in the list, so it must be *with*.

Compared with using a decoding tree, canonical coding is very direct. An explicit decode tree of the kind illustrated in Figure 2.5 requires a lot of space and is accessed randomly. With the canonical representation, only the first codeword of each length is accessed, plus one access to a lookup table to determine what the word is. Storing the first codeword of each length takes negligible space—the example in Table 2.2 has just 14 different lengths, ranging from 4 to 17. The list of symbols in lexical order of their codewords replaces the randomly accessed Huffman tree and is consulted only once for each symbol decoded.

Canonical Huffman codes

We now take a more detailed look at the implementation of a Huffman encoder and decoder pair. These details are quite intricate and can be skipped on the first reading, which is why this section is marked "optional" with a gray bar in the margin.

A canonical code is carefully structured to allow extremely fast decoding, with a memory requirement of only a few bytes per alphabet symbol. The code is called "canonical" (standardized) because much of the nondeterminism of normal Huffman codes is avoided. For example, in normal construction of the Huffman tree, some convention such as using a 0 bit to indicate a left branch and a 1 bit to indicate a right is assumed, and different choices lead to different, but equally valid, codeword assignments.

It is easiest to show this effect with an example. Consider the information shown in Table 2.3. The second column shows the observed symbol frequencies for the symbols in column 1. Three possible prefix-free codes for these symbols are shown in the columns headed *Code 1*, *Code 2*, and *Code 3*. Of course, in word-based codes there would be thousands of symbols instead of the six shown here, with frequencies ranging from 1 for *hapax legomena* to many thousands for common words (like *the*)—just as in Table 2.2.

The derivation of the first two codes in Table 2.3 is exactly as described in Figure 2.6. At each step, the two smallest items are extracted and coalesced, with one of the items having all its codewords prefixed by a 0 bit and the symbols represented by the other item being prefixed by a 1 bit. To obtain Code 1, for example, the sideways tree in Figure 2.8 was used, in which the convention is adopted that the upper

tions (Moffat and Turpin 1998). The combining rule—which requires aggregating the smallest two frequencies—becomes rather more complex, but a substantial reduction in time and space is possible, provided only that the list of input symbol frequencies is already in sorted order. Run-length coding can also be applied to the output (such as the lengths shown in Table 2.2), which also inevitably contains long runs of identical code lengths. Analysis of the run-length method for calculating Huffman codes shows that the time taken is proportional to $r + r \log(n/r)$ for an alphabet of n symbols in which there are r distinct symbol frequencies (Moffat and Turpin 1998). Furthermore, the space required is proprtional to the running time. For typical large-alphabet values n = 1,000,000 and r = 10,000, these bounds represent a considerable saving in resources, and the mechanism is very useful in situations when the $n \log n$ cost of sorting the symbol frequencies can be either avoided or amortized over more than one calculation.

Summary

Now, at last, we have seen how to generate and use Huffman codes efficiently for alphabets containing thousands of symbols. This is an essential prerequisite for virtually all large-scale information retrieval systems that use compression. Although Huffman codes are standard fodder for undergraduate courses in computer science, the methods described in the usual textbooks do not scale up effectively. Considerable savings can be made in both the time and the main memory required for decoding by using canonical Huffman codes instead of the standard decoding tree, and decoding is one of the principal ongoing operations in compressed information retrieval systems. Although encoding is done far less often—just once every time the database is reconstructed—it is still necessary to accomplish it within reasonable resource constraints, and again substantial savings in both time and space can be reaped by using the nonstandard, and by no means obvious, techniques that we have described.

2.4 Arithmetic coding

Arithmetic coding is a technique that has made it possible to obtain excellent compression using sophisticated models. Its principal strength is that it can code arbitrarily close to the entropy. It has been shown that it is not possible to code better than the entropy on average (Shannon 1948), so in this sense arithmetic coding is optimal.

To compare arithmetic coding with Huffman coding, suppose symbols from a binary alphabet are to be coded, where the symbols have probabilities of 0.99 and 0.01. The information content of a symbol s with probability $\Pr[s]$ is $-\log \Pr[s]$ bits, so the symbol with probability 99 percent can be represented using arithmetic coding in just under 0.015 bits. In contrast, a Huffman coder must use at least one bit per symbol. The only way to prevent this situation with Huffman coding is to "block" several symbols together at a time. A block is treated as the symbol to be coded, so that the per-symbol inefficiency is now spread over the whole block.

Blocking is difficult to implement because there must be a block for every possible combination of symbols, so the number of blocks increases exponentially with their length; this effect is exacerbated if consecutive symbols come from different alphabets, as is the case in the high-performance schemes that are described in the next few sections.

The compression advantage of arithmetic coding is most apparent in situations like the previous example, in which one symbol has a particularly high probability. More generally, it has been shown that the inefficiency of Huffman coding is bounded above by

$$\Pr[s_1] + \log \frac{2 \log e}{e} \approx \Pr[s_1] + 0.086$$

bits per symbol, where s_1 is the most likely symbol (Gallager 1978). For the extreme example described above, the inefficiency is thus 1.076 bits per symbol at most (a bound that can trivially be reduced to one bit per symbol), which is many times higher than the entropy of the distribution. On the other hand, for English text compressed using a zero-order character-level model, the entropy is about five bits per character, and the most common character is usually the space character, with a probability of about 0.18. With such a model the inefficiency is less than $0.266/5 \approx 5.3$ percent. In many compression applications $\Pr[s_1]$ is even smaller, and the inefficiency becomes almost negligible. Moreover, this is an *upper* bound, and in practice the inefficiency is often significantly less than the bound might indicate. On the other hand, when images are being compressed, it is common to deal with two-symbol alphabets and highly skewed probabilities, and in these cases arithmetic coding is essential unless the symbol alphabet is altered using a technique like blocking.

Another advantage of arithmetic coding over Huffman coding is that arithmetic coding calculates the representation on the fly, so less primary memory is required for operation and adaptation is more readily accommodated. Canonical Huffman codes are also fast, but they are suitable only if static or semi-static modeling is being used. Arithmetic coding is particularly suitable as the coder for high-performance adaptive models, where very high probabilities (confident predictions) are occurring, where many different probability distributions are stored in the model, and where each symbol is coded as the culmination of a sequence of lower-level decisions.

One disadvantage of arithmetic coding is that it is slower than Huffman coding, especially in static or semi-static applications. Also, the nature of the output means that it is not easy to start decoding in the middle of a compressed stream. This contrasts with Huffman coding, in which it is possible to index "starting points" in a compressed text if the model is static. In full-text retrieval, both speed and random access are important, so arithmetic coding is not likely to be appropriate. Furthermore, for the types of models used to compress full-text systems, Huffman coding gives compression that is practically as good as arithmetic coding. Thus, in large collections of text and images, Huffman coding is likely to be used for the text and arithmetic coding for the images.

How arithmetic coding works

Arithmetic coding can be somewhat difficult to grasp. Though it is interesting, understanding it is not essential. Source code is readily available for efficient arithmetic coders (see *www.cs.mu.oz.au/mg/*), which can be plugged into a compression system to perform the coding part. Only the interface to the coder needs to be understood in order to use it.

The output of an arithmetic coder, like that of any other coder, is a stream of bits. However, it is easier to describe arithmetic coding if we prefix the stream of bits with 0. and think of the output as a fractional binary number between 0 and 1. In the following example, the output stream is 1010001111, but it will be treated as the binary fraction 0.1010001111. In fact, for the sake of readability, the number will be shown as a decimal value (0.64) rather than as binary, and some possible efficiencies will be overlooked initially.

As an example we will compress the string *bccb* from the ternary alphabet $\{a, b, c\}$. We will use an adaptive zero-order model and deal with the zero-frequency problem by initializing all character counts to one.

When the first b is to be coded, all three symbols have an estimated probability of $1/3$. An arithmetic coder stores two numbers, *low* and *high*, which represent a subinterval of the range 0 to 1. Initially, $low = 0$ and $high = 1$. The range between *low* and *high* is divided according to the probability distribution about to be coded. Figure 2.18a shows the initial interval, from 0 to 1, with a third of it allocated to each symbol. The arithmetic coding step simply involves narrowing the interval to the one corresponding to the character to be coded. Thus, because the first symbol is a b, the new values are $low = 0.3333$ and $high = 0.6667$ (working to four decimal places). Before coding the second character, c, the probability distribution adapts because of the b that has already been seen, so $\Pr[a] = 1/4, \Pr[b] = 2/4$, and $\Pr[c] = 1/4$. The new interval is now divided up in these proportions, as shown in Figure 2.18b, and coding of the c involves changing the interval so that it is from $low = 0.5834$ to $high = 0.6667$. Coding continues as shown in Figure 2.18c, and at the end the interval extends from $low = 0.6390$ to $high = 0.6501$.

At this point, the encoder transmits the code by sending any value in the range from *low* to *high*. In the example, the value 0.64 would be suitable. The decoder simulates what the encoder must have been doing. It begins with $low = 0$ and $high = 1$ and divides the interval as shown in Figure 2.18a. The transmitted number, 0.64, falls in the part of the range corresponding to the symbol b, so b must have been the first input symbol. The decoder then calculates that the range should be changed to $low = 0.3333$ and $high = 0.6667$, and, because the first symbol is now known to be b, the new probability allocation where $\Pr[b] = 2/4$ (shown in Figure 2.18b) can be calculated. Decoding proceeds along these lines until the entire string has been reconstructed.

A general algorithm for calculating the range during encoding is shown in Figure 2.19, and Figure 2.20 shows how a symbol is decoded and the range is updated afterward. Since arithmetic coding deals with ranges of probabilities, it is usual for a model to supply *cumulative probabilities* to the encoder and decoder; this makes

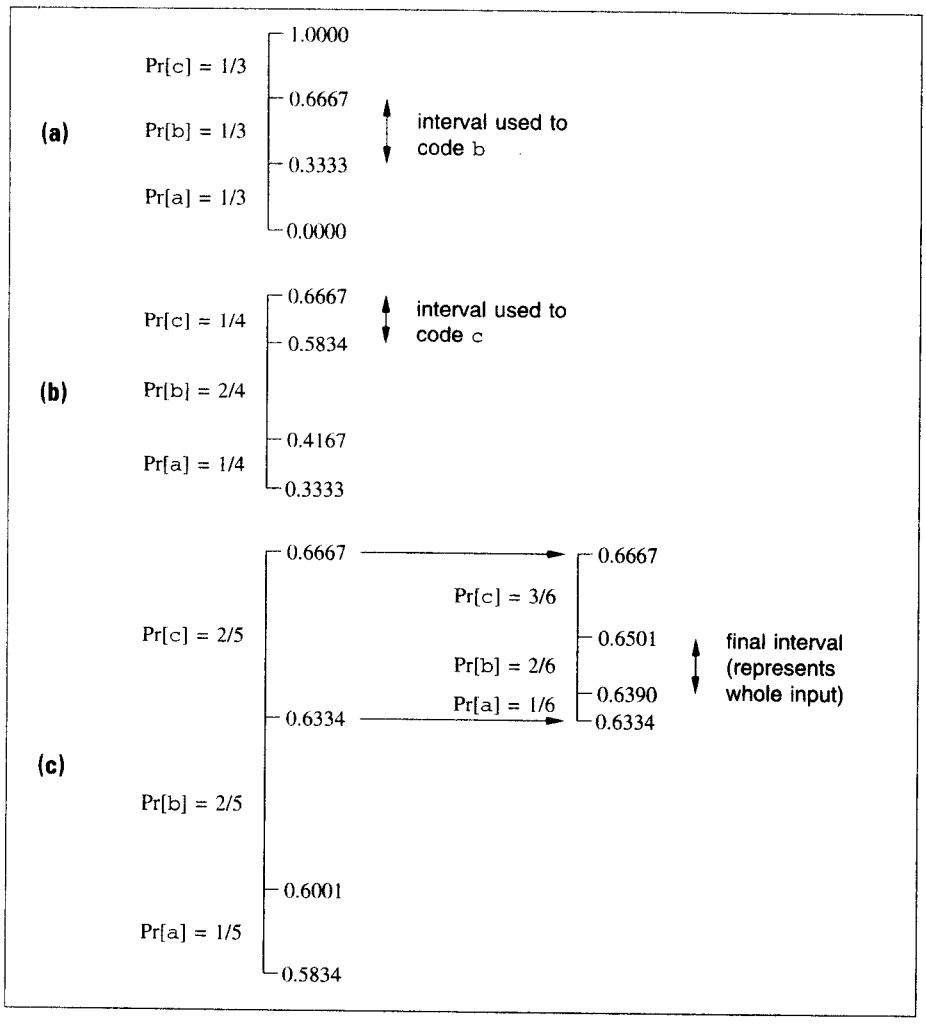

Figure 2.18 Arithmetic coding example for the string *bccb*: (a) first symbol; (b) second symbol; (c) third and fourth symbols.

the first steps of each algorithm easy to implement. For static and semi-static coding, the cumulative probabilities can be stored in an array. The encoder accesses the array by symbol number, and the decoder accesses it by binary search or through the use of a lookup table. Adaptive coding requires a more sophisticated structure so that cumulative probabilities can be adjusted on the fly. More commonly, frequency counts are kept for each symbol, and these are normalized to probabilities as an integral part of the arithmetic coding process.

Informally, compression is achieved because high-probability events do not decrease the interval from *low* to *high* very much, while low-probability events result in a much smaller next interval. A small final interval requires many digits (bits)

To code symbol s, where it is assumed that symbols are numbered from 1 to n, and that symbol i has probability $\Pr[i]$,

1. Set *low_bound* $\leftarrow \sum_{i=1}^{s-1} \Pr[i]$.
2. Set *high_bound* $\leftarrow \sum_{i=1}^{s} \Pr[i]$.
3. Set *range* $\leftarrow$ *high* $-$ *low*.
4. Set *high* $\leftarrow$ *low* + *range* $\times$ *high_bound*.
5. Set *low* $\leftarrow$ *low* + *range* $\times$ *low_bound*.

Figure 2.19 The arithmetic encoding step.

To decode a symbol, assuming that the symbols are numbered from 1 to n, that symbol i has probability $\Pr[i]$, and that *value* is the arithmetic code to be processed,

1. Find s such that
$$\sum_{i=1}^{s-1} \Pr[i] \leq (value - low)/(high - low) < \sum_{i=1}^{s} \Pr[i].$$
2. Perform the range-narrowing steps described in Figure 2.19, exactly as if symbol s is being *en*coded.
3. Return symbol s.

Figure 2.20 The arithmetic decoding step.

to specify a number that is guaranteed to be within the interval—for example, at least six decimal digits are needed to specify a number that is between 0.378232 and 0.378238. In contrast, a large interval requires few digits—for example, any number beginning with 0.4 is between 0.378232 and 0.578238, so the decoder only needs to know that the number begins with 0.4. The number of digits necessary is proportional to the negative logarithm of the size of the interval, just as the number of digits needed to represent numbers on the other side of the decimal point is proportional to the positive logarithm of the number. The size of the final interval is the product of the probabilities of the symbols coded, and so the logarithm of this quantity is the same as the sum of the logs of the individual probabilities. Therefore, a symbol s of probability $\Pr[s]$ contributes $-\log \Pr[s]$ bits to the output, which is equal to the symbol's information content and results in a code that is identical to the bound given by the entropy formula. This is why arithmetic coding produces a near-optimal number of output bits and is, in effect, capable of coding

high-probability symbols in just a fraction of a bit. In practice, arithmetic coding is not exactly optimal because of the use of limited precision arithmetic and because a whole number of bits (or even bytes) must eventually be transmitted, but it is extremely close.

As we have described arithmetic coding so far, nothing appears in the output until all the encoding has been completed. In practice, it is possible to output bits *during* encoding, which avoids having to work with higher and higher precision numbers. The trick is to observe that when *low* and *high* are close in value, they have a common prefix. For example, after the third character has been coded (Figure 2.18c), the range is *low* = 0.6334 and *high* = 0.6667. Both are prefixed with 0.6, so no matter what happens to the range from there on, the first symbol transmitted must be a 6 (or would be, if the encoder were working in decimal). Thus, an output symbol can be transmitted at this point and then removed from *low* and *high* without any effect on the remaining calculations. In the example, the first decimal digit (which is 6) can be removed from *low* and *high*, changing them to 0.334 and 0.667, respectively. Working with these new values, the top of the *a* interval is now calculated to be 0.390 instead of 0.6390. The same output is being constructed, but the need for increasing precision has been avoided, and the output is generated incrementally, rather than having to wait until the end of coding.

As mentioned earlier, the final output value is really transmitted as a binary fractional number, and the 0. on the front is unnecessary because the decoder knows that it will appear. The values *low* and *high* are stored in binary; in fact, with suitable scaling they can be stored as integers rather than as real numbers. Working with finite precision causes compression to be a little worse than the entropy bound, but 16- or 32-bit precision usually degrades compression by an insignificant amount. Witten, Neal, and Cleary (1987), in a seminal paper, describe a general-purpose arithmetic coder based upon the use of integer arithmetic, and they also give analysis to bound the inefficiency arising from calculation errors. So although the example used unlimited-precision floating-point decimal numbers, in practice arithmetic coding can be implemented using fixed-precision integers, and the output is a stream of bits. Each symbol coded requires just a few arithmetic operations in the arithmetic coder. In the following section, we look more carefully at exactly how many operations are necessary.

Implementing arithmetic coding

This section describes a practical implementation of arithmetic coding. As well as examining how the interface between the model and coder works in practice, we will look at the special cases where the alphabet is very small (perhaps just two symbols) or very large (thousands or millions of symbols).

The interface between an arithmetic coder and a model is a little unusual because of the way that arithmetic coding works. To divide up the range, the coder needs to know which part of it corresponds to the symbol to be coded. We now describe three routines that are found in the mg source code: *arithmetic_encode*, *arithmetic_decode_target*, and *arithmetic_decode*.